Vyhledávání ve vyvážených stromech II.

Jan M Honzík

Abstrakt

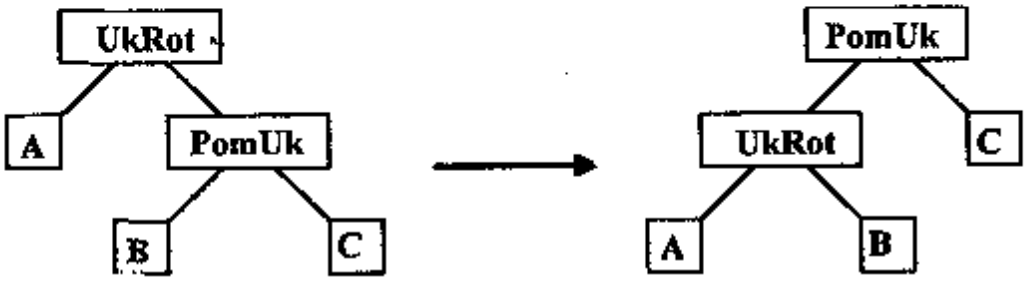
Příspěvek navazuje na stejnojmenný referát ze sborník konference Programování '93, [Hon93] který se zabýval tzv. 2-3-4 stromy a "Červeno-černými" stromy. V tomto příspěvku je poněkud jiný pohled na "červeno-černé" (RB) stromy. V tomto pohledu se "barví" uzly a ne hrany, používají se jednoduché rotace, je uvedena jiná filozofie operací a je uvedena i operace Delete. Dále je uveden další typ vyhledávacího stromu, tzv. Splay-Tree, který se vyvažuje po každé operaci i po vyhledávání. Často referované uzly se hromadí poblíž kořene stromu, čímž se statisticky snižuje doba jejich vyhledávání, která má nejhorší případ opět O(lg n).

1. Jiný pohled na Červeno-Černé stromy

Poněkud jiný přístup k RB stromům je uveden v jedné z posledních nejvyspělejších publikací [Cor]. RB strom je uveden bez návaznosti k 2- 3-4 stromům, místo hran se "barví" uzly, a strom je definován takto:
Binární strom je RB strom, když má všechny čtyři následující vlastnosti:

- Každý uzel je buď červený nebo černý. (V obrázcích písmeno "B" pro "black" a "R" pro "red".)
- Každý terminální uzel, za nějž se považuje prázdný uzel "nil" je černý.
- Je-li uzel červený, pak jsou oba jeho synovské uzly černé.
- Cesty z každého uzlu ke všem terminálním uzlům tohoto uzlu obsahují stejný počet černých uzlů.

Při levé rotaci uzlu x předpokládáme, že pravý syn uzlu je neprázdný.



```
procedure LeftRot(Kor, UkRot:TUk);
var
 PomUk:TUk;
begin
 PomUk:=UkRot^.PUk; (* uchování pravého syna rotujiciho uzlu v PomUk *)
 UkRot^.PUk:=PomUk^.LUk; (* Pravým synem se stane levý
                    syn pravého syna *)
 if PomUk^.LUk <> nil
 then PomUk^.LUk^.ZUk:=UkRot;
 PomUk^.ZUk:=UkRot^.ZUk; (* připojení rodiče UkUz k PomUk *)
 if PomUk^.ZUk = nil
 then Kor^.Root:=PomUk
 else if UkRot=UkRot^.ZUk^.LUk
    then UkRot^.ZUk^.LUk:=PomUk
    else UkRot^.ZUk^.PUk:=PomUk;
 PomUk^.LUk:=UkRot;
 UkRot^.ZUk:=PomUk
end;
```

Operace Insert v RB stromu.

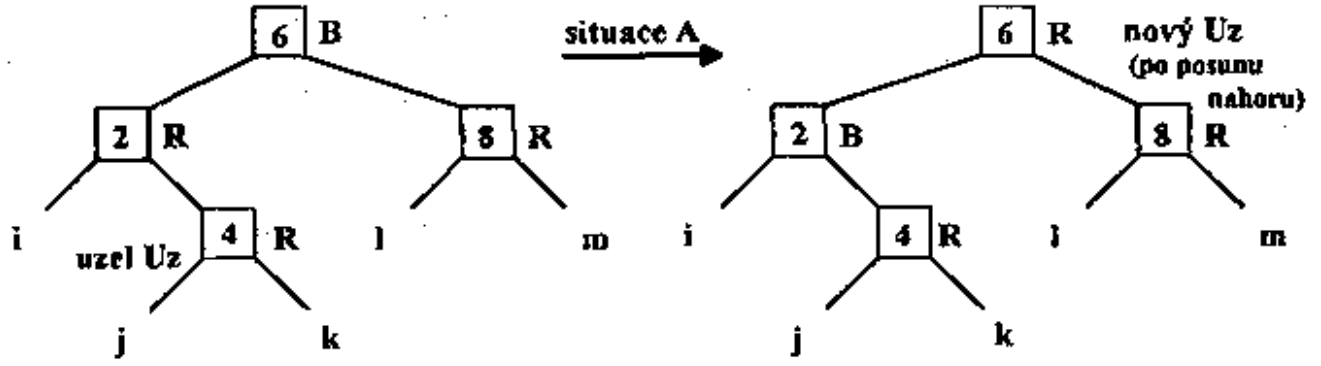
Součástí operace Insert do RB stromu je vložení uzlu do stromu, jako do prostého binárního stromu. Tuto běžně známou operaci popisuje následujici procedura, jejimiž parametry jsou hlavičkový uzel, ukazujíci na kořen stromu, a ukazatel na naplněný uzel, jehož oba ukazatelé dolů jsou nilové a interpretují se jako prázdné a černé uzly. (prázdný strom sestává z hlavičkového uzlu, jehož jediný funkční ukazatel, označený dále jako složka KOREN, je nulový), a. procedure BinTreeInsert(Kor, Uz:TUk); (* Uzel Uz obsahuje vkládaný klíč a ukazatele na levý i pravý podstrom jsou nilové; Uzel KOR je hlavičkou stromu, která svou sloužkou Koren, ukazuje na kořen stromu; v praxi je výhodnější použít záznam s variantou, která rozlišuje záznam uzlu hlavičky od jiných uzlů, nebo pro případ hlavičky použít jako ukazatel na kořen levý nebo pravý ukazatel *)

```
var
 PomUk1, PomUk2:TUk;
begin
 PomUk2:=nil;
 PomUk1:= Kor^.Koren;
 while PomUk1 <> nil do begin
  if Uz^.Key<PomUk1^.Key
  then x:= PomUk1^.LUk
  else x:= PomUk1^.PUk;
 end (* while *);
 Uz^.ZUk:=PomUk1
 if PomUk2=nil
 then Kor^.Koren:=Uz
 else if Uz^.Key < PomUk2^.Key
    then PomUk2^.LUk:=Uz
    else PomUk2^.PUk:=Uz
end;
```

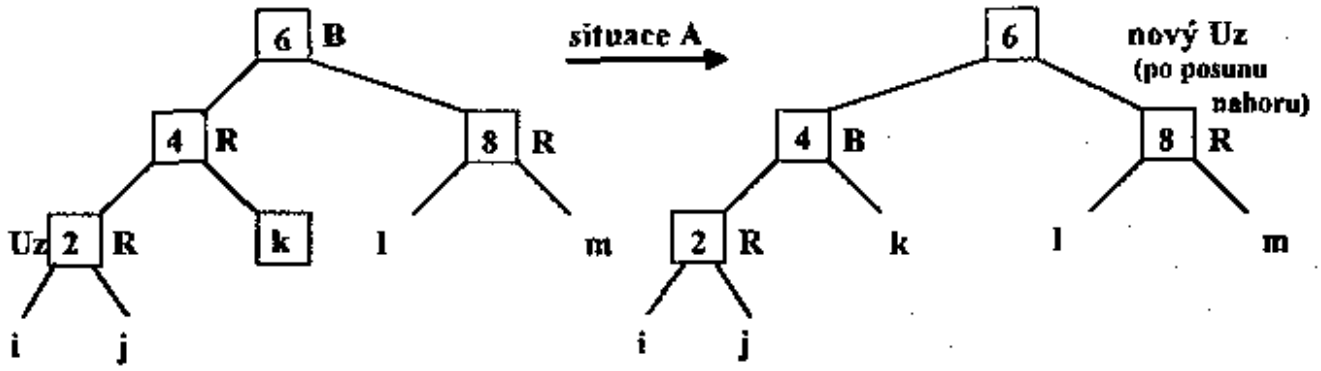
```
procedure RedBlackTreeInsert(Kor,Uz:TUk);
(* Uzel obsahuje vkládaný klíč a jeho ukazatelé vlevo
a vpravo jsou nilové *)

var
 PomUk:TUk;
begin
 BinTreeInsert(Kor, Uz);
 Uz^.Red:=true;
 while (Uz <>Kor^.Koren) and Uz^.ZUk^.Red do begin
  if Uz^.ZUk= Uz^.ZUk^.ZUk^.LUk
  then begin (* otec uzlu Uz je levým synem *)
   PomUk:=Uz^.ZUk^.ZUk^.PUk;
   if PomUk^.Red
   then begin (* Situace A *)
    Uz^.ZUk^.Red:=false;
    PomUk^.Red:=false;
    Uz^.ZUk^.ZUk^.Red:=true;
    Uz:=Uz^.ZUk^.ZUk
   end else begin
    if Uz=Uz^.ZUk^.PUk
    then begin (* Situace B *)
     Uz:=Uz^.ZUk;
     LeftRot(Kor,Uz);
    end;

    Uz^.ZUk^.Red:=false; (* Situace C *)
    Uz^.ZUk^.ZUk^.Red:= true;
    RightRed(Kor,Uz^.ZUk^.ZUk);
   end
  end else begin (* otec uzlu Uz je pravým synem *)

   (* symetrická varianta úseku za then; dojde k výměně LUk<=>ZUk *)
  end;
 Kor^.Koren^.RedL:=false;
end;
```

Dva příkazy před cyklem while mohou být příčinou porušení vlastnosti RB stromu. Jedinou porušitelnou vlastností je vlastnost č.3, která říká, že je-li uzel červený, pak jsou oba jeho synovské uzly černé. Je-li vložený uzel Uz i jeho otec červený a "strýc" je také červený, nastává situace "A". Je-li Uz i otec červený, ale strýc je černý, pak je-li Uz pravým uzlem, nastává situace "B". Provede se levá rotace. Uz se stane levým uzlem a nastává situace 3. Další pravá rotace upraví situaci v okolí uzlu. Další průchod cyklem nastává, pokud se podmínky opakují "ve vyšším patru". Situace 1 se od situace 2 a 3 liší barvou "strýce", která je pro případ 2 a 3 černá. Cyklus posouvá problém rotací směrem nahoru a končí, dosáhne-li kořene, nebo nastane-li situace 2 nebo 3, které cyklus ukončují.

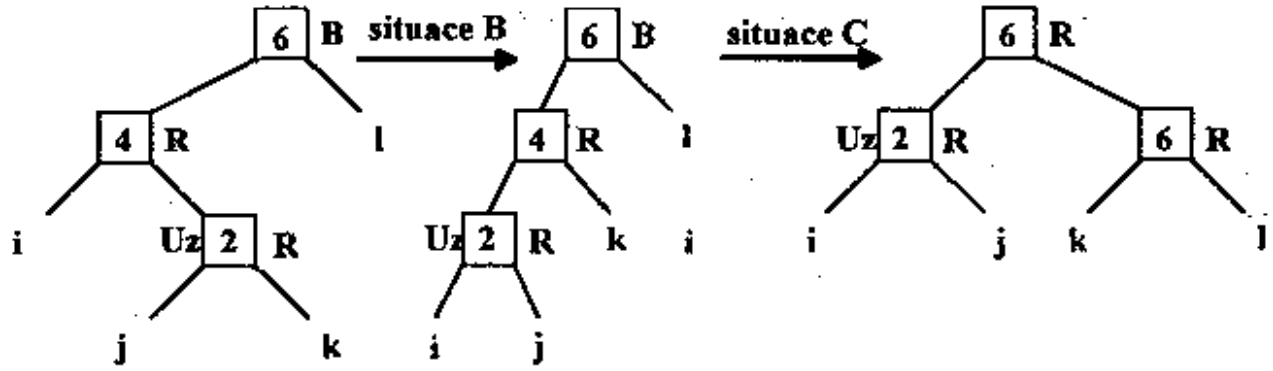
Protože výška RB stromu je O(ln n) je i časová složitost vložení nového uzlu O(ln n); cyklus while se opakuje jen v případě situace 1 a pak se ukazatel posunuje směrem nahoru. Maximální počet opakování cyklu je tedy opět O(ln n). Celkem je tedy složitost vkládání uzlu do RB stromu O(ln n). Ve skutečnosti se však nikdy neprovede více než 2 rotace, protože po rotaci v situaci A nebo B cyklus končí.



Bylo porušeno pravidlo 3, protože Uz a jeho rodič byli červení.



Bylo porušeno pravidlo 3, protože Uz a jeho rodič byli červení.



Operace Delete v RB stromu.

Pro zjednodušení koncových podmínek cyklu zavedeme do stromu zarážku ve formě zvláštního uzlu, který bude sloužit jako nil. Na tento "nilový" uzel ukazuje ukazatel Kor^.nil. Má strukturu jako každý jiný uzel a jeho složky jsou libovolné. Účelem je možnost použití "nilového" uzlu jako by to byl normální uzel, jehož otec je uzel Uz. Bylo by samozřejmě možné vytvořit místo každého nilu nový "nilový" uzel, ale to by byla zbytečná ztráta paměti. Stačí jeden nilový uzel, na který ukazují všechny ukazatelé, kteří v předchozím případě byly nilové. Budeme-li však kdekoli manipulovat se synovským uzlem uzlu Uz, musíme napřed nastavit Strom^.Nil^.ZUk. Procedura

RedBlackTreeDelete je modifikací klasického TreeDelete pro binární strom. Po vypojení rušeného uzlu Uz se vyvolá rekonfigurační procedura, která provede rotace a znovuustaví vlastnosti RB stromu. Klasická operace BinaryTreeDelete má následující tvar: function BinaryTreeDelete(Kor,Uz:TUk):TUk; (* Kor je ukazatel na hlavičku stromu, Uz je ukazatel na rušený uzel; funkce vrací uzel učený ke zrušení - což nemusí být fyzicky týž uzel, na který ukazoval na počátku ukazatel Uz *)

```
var
 PomUk1, PomUk2:TUk; (* Pomocné ukazatele *)
begin
 if (Uz^.LUk = nil) or (Uz^.PUk = nil)
 then PomUk2:=Uz
 else PomUk2:=TreeSuccessor(Uz); (* uzel za "Uz" v průchodu inorder *)

 if PomUk2^.LUk<>nil
 then PomUk1:=PomUk2^.LUk
 else PomUk1:=PomUk2^.PUk;

 if PomUk1 <> nil (* A *)
 then PomUk1^.ZUk := PomUk2^.ZUk;

 if PomUk2^.ZUk = nil
 then Kor^.Koren:=PomUk1
 else if PomUk2:=PomUk2^.ZUk^.LUk
    then PomUk2^.ZUk^.LUk:=PomUk1
    else PomUk2^.ZUk^.PUk:=PomUk1;

 if PomUk2 <> Uz
 then Uz^.Key:=PomUk2^.Key; (* je-li v uzlu více složek, pak
                               kopírování všech *)
 BinaryTreeDelete:=PomUk2; (* vrací se ukazatel na uzel ke
                              zrušení *)
end;
```

Funkce TreeSuccessor určí následníka ve stromu ve smyslu průchodu inorder (tedy uzel s hodnotou nejblíže vyšší).

```
function TreeSuccessor(Uz:TUk):TUk;
var
 PomUk:TUk;
begin
 if Uz^.PUk<>nil
 then begin (* má pravý podstrom; následník je nejlevější na diagonále
               pravého podstromu *)
  while Uz^.LUk<>nil do Uz:=Uz^.LUk; (* Nejlevější na diagonále *)
  TreesSuccessor:=Uz;
 end else begin (* nemá pravý podstrom; existuje-li, pak následník je
                   nejnižší předek, který má levé dítě, jež je také
                   předkem daného uzlu *)
  PomUk:=Uz^.ZUk;
```

```
   while (PomUk<>nil) and (Uz=PomUk^.PUk) do begin
    Uz:=PomUk; (* synovský uzel *)
    PomUk:=PomUk^.ZUk (* otcovský uzel *)
   end;
   TreeSuccessor:= PomUk;
  end; (* if Uz^.PUk<>nil *)
end;



function RedBlackTreeDelete(Kor, Uz:Tuk):TUk;
(* Kor je ukazatel na hlavičku stromu, Uz je ukazatel na rušený uzel;
   funkce vrací uzel učený ke zrušení - což nemusí být fyzicky týž uzel,
   na který ukazoval na počátku ukazatel Uz *)


var
  PomUk1, PomUk2:TUk;
begin
  if (Uz^.LUk = Kor^.Nil) or (Uz^.PUk = Kor^.Nil)
  then PomUk2:=Uz
  else PomUk2:= TreeSuccessor(Uz);(* uzel za "Uz" v průchodu inorder *)

  if PomUk2^.LUk <> Kor^.Nil
  then PomUk1:=PomUk2^.LUk
  else PomUk1:=PomUk2^.PUk;

  PomUk1^.ZUk:=PomUk2^.ZUk; (* A - oproti BinTreeDel bez podmínky *)

  if PomUk2^.ZUk = Kor^.Nil
  then Kor^.Koren:= PomUk1
  else if PomUk2=PomUk2^.ZUk^.LUk
      then PomUk2^.ZUk^.LUk:=PomUk1
      else PomUk2^.ZUk^.PUk:=PomUk1;

  if PomUk2<>Uz
  then Uz^.Key:=PomUk2^.Key; (* je-li v uzlu více složek, pak
                                kopírování všech *)

  if not PomUk2^.Red
  then ReconfigReBlacktree(Kor,PomUk1);

  RedBlackTreeDelete := PomUk2
end;
```

RedBlackTreeDelete se liší od BinaryTree Delete použitím pomocného nilu, absencí podmínky (viz komentář (* A *)) a aplikací rekonfiguračního mechanismu. Rekonfigurační mechanismus popisuje následující procedura:

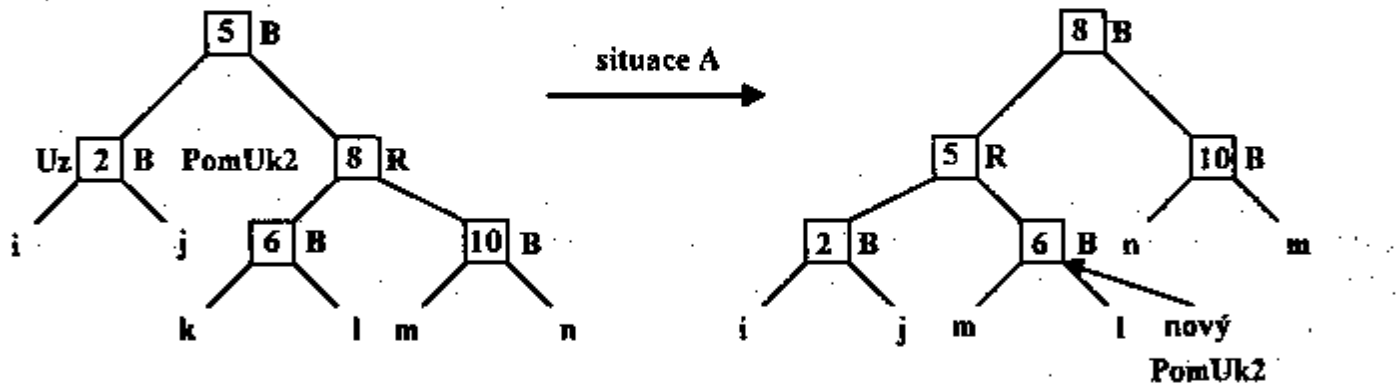
```
procedure ReconfigReBlackTree(Kor,Uz:Tuk);
var
 PomUk2:TUk;
begin
 while (Uz<> Kor^.Koren) and (not Uz^.Red) do begin
  if Uz=Uz^.ZUk^.LUk
  then begin (* uzel je levým synem *)
   PomUk:=Uz^.ZUk^.PUk; (* PomUk je pravý strýc *)
   if PomUk2^.Red (* Pravý strýc je červený *)
   then begin
    PomUk2^.Red:= false (* Situace A *)
    Uz^.ZUk^.Red:=true; (* Situace A *)
    LeftRot(Kor,Uz^.ZUk); (* Situace A *)
    PomUk2:=Uz^.ZUk^.PUk; (* Situace A *)
   end;

   if not PomUk2^.LUk^.Red and not PomUk2^.PUk^.Red
   then begin (* oba synové PomUk2 jsou černí *)
    PomUk2^.Red:=true; (* Situace B *)
    Uz:=Uz^.ZUk (* Situace B *)
   end else begin
    if not PomUk2^.PUk^.Red
    then begin
     PomUk2^.LUk^.Red:= false; (* Situace C *)
     PomUk2^.Red:=true; (* Situace C *)
     RightRot(Kor,PomUk2); (* Situace C *)
     PomUk2:=Uz^.ZUk^.PUk; (* Situace C *)
    end;

    PomUk2^.Red:=Uz^.ZUk^.Red; (* Situace D *)
    Uz^.ZUk^.Red:=false; (* Situace D *)
    PomUk2^.PUk^.Red:=false; (* Situace D *)
    LeftRot(Kor,Uz^.ZUk); (* Situace D *)
    Uz:=Kor^.Koren (* Situace D *)
   end else begin (* uzel je levým synem *)
     (* symetrická sekvence se záměnou LUk <=> PUk *)
   end;
  end; (* if *)
 end; (* while *)

 Uz^.Red:=false;
end; (* procedure *)
```

Je-li fyzicky vyřazený uzel černý, způsobí, že na větvi je o jeden černý uzel méně, čímž se poruší pravidlo 4. Tento případ lze korigovat úvahou, že uzel Uz má o jednu černou barvu "navíc". Jinými slovy, připočteme-li ke každé cestě obsahující uzel Uz o jeden černý uzel navíc, zachováme platnost pravidla 4. Když vyřadíme černý uzel, spustíme jeho duplicitní černost na potomka. Tím je ale porušeno pravidlo 1, které nedovoluje dvojitost barvy.

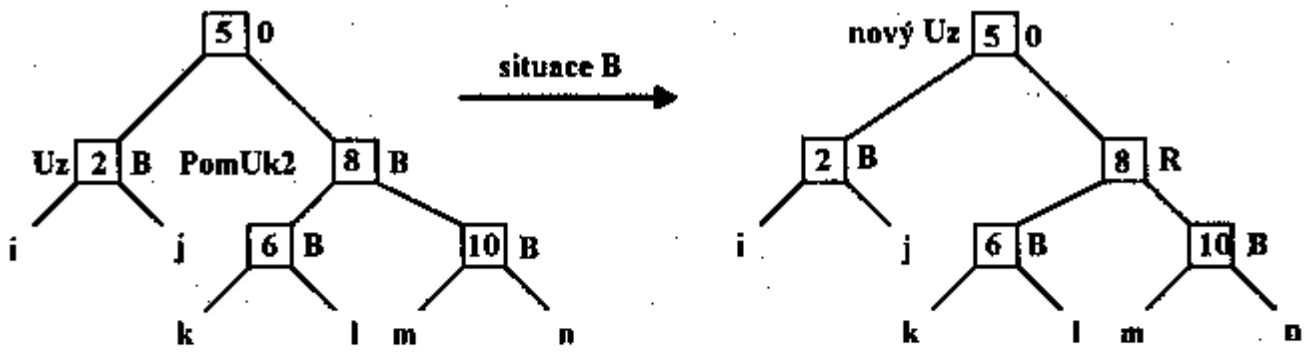
Procedura ReconfigRedBlackTree znovuustavuje pravidlo 1. Účelem cyklu while je posunování nadbytečného černého uzlu směrem nahoru pokud je Uz různý od kořene - v tom případě lze snadno

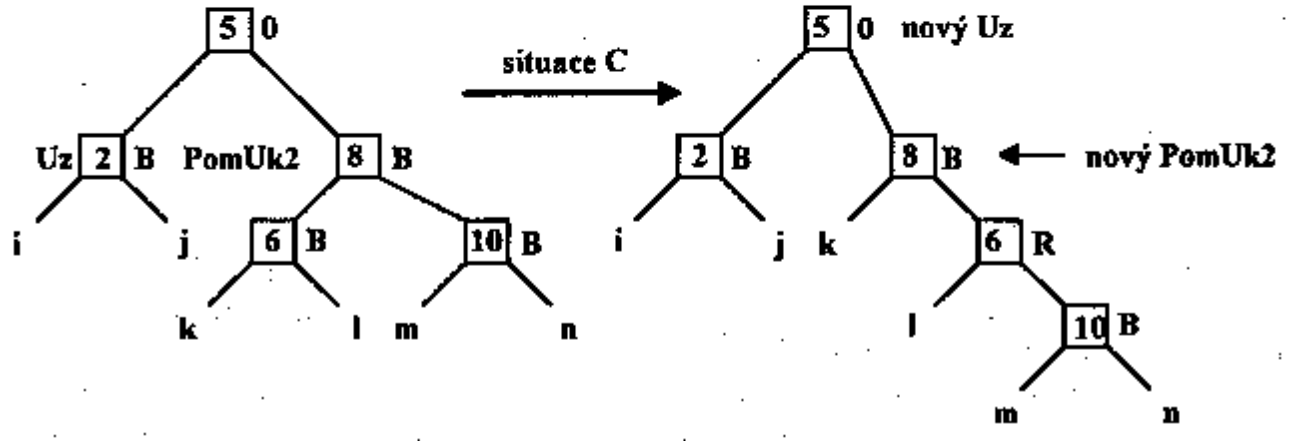
odstranit - nebo dokud není červený - v tom případě je na posledním řádku přebarven na černý. Pokud tyto podmínky nenastanou, provádí se v cyklu odpovídající rotace a přebarvování.

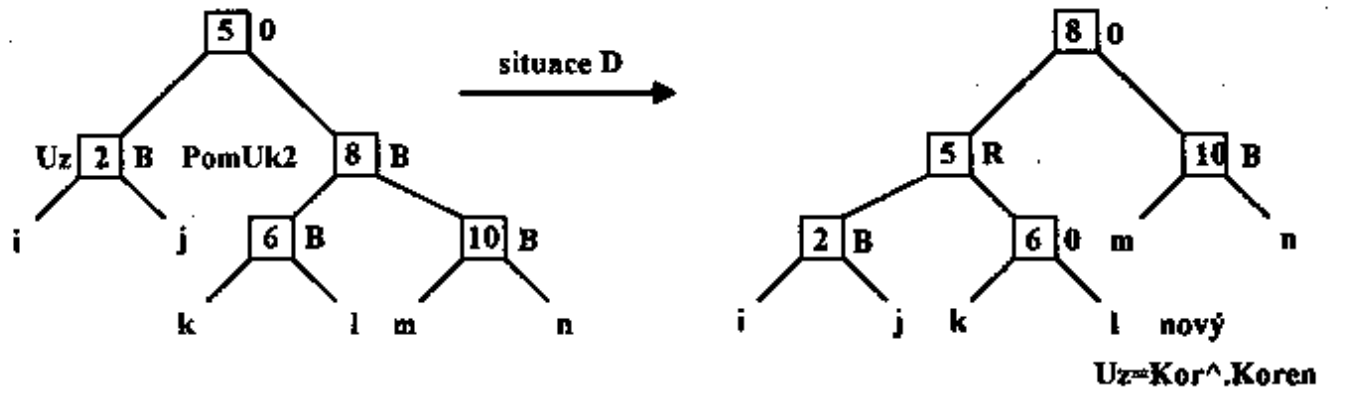
V následujících obrázcích je naznačena rotace a přebarvení podle situací naznačených v komentáři procedury ReconfigRedBlackTree.



V následujících obrázcích je písmenem O označen uzel, který může být jak červený, tak černý.







Algoritmus RedBlackTreeDelete má bez rekonfigurace složitost O(lg n), protože výška stromu je O(lg n). V proceduře ReconfigRedBlackTree se v situacích A,C a D provede konstantní počet přebarvení a maximálně 3 rotace. Pouze v situaci B se cyklus while může opakovat, a ukazatel Uz se posunuje nahoru maximálně O(lg n) krát bez rotací. Celkově je tedy složitost rekonfigurace O(lg n) a celkového rušení také O(ln n).

2. Hodnocení červeno-černých stromů

Shrnutím všech uvedených algoritmů získáme vyhledávací mechanismus s vkládáním, který zaručuje logaritmickou složitost vyhledávání i vkládání. Přesná analýza pro průměrnou délku vyhledání nebyla zatím provedena, ale výsledky částečných analýz a experimentální výsledky jsou přesvědčivé. Nejvýznamnější jsou vlastnosti RB stromů v nejhorším případě a nízká časová složitost vkládání s vyvážením. Starším mechanismem pro vyváženost vyhledávání ve stromech je založen na tzv. AVL stromech, pro něž platí, že výšky podstromů každého uzlu AVL stromu se liší maximálně o 1. Poruší-li se toto pravidlo při vkládání nebo rušení, provádí se rekonstrukce vyváženosti prostřednictvím rotací. Vyvážení ale vyžaduje další cyklus. Vkládání se provádí tak, že po vložení se algoritmus vrací po cestě zpět a rotacemi znovuustavuje rovnováhu. U těchto stromů je také nutné tříbitová informace v každém uzlu o tom, zda je uzel vlevo lehčí, vyvážený nebo vpravo těžší. Jiným známým mechanismem jsou 2-3 stromy, obsahující pouze 2 a 3- uzly. Operace vkládání má však také cyklus navíc včetně rotací, podobně jako u AVL stromů. "Červeno-černý" mechanismus náhrady "3-uzlů" těchto stromů by situaci pomohl. 2-3-4-stromy jsou však výhodnější, protože vyhledávají shora dolů a snaží se vložit na terminálních uzlech stromu, pokud to není 4-uzel. Metoda 2-3 stromů má výhodu v tom, že se na jedno vkládání provádí jen jedna rotace, což může být u některých aplikací výhodné. Implementace této metody je však poněkud složitější, než u "top-down" metody 2-3-4-stromů, popsané v této kapitole.

3. SPLAY TREE

Splay strom (splay tree, rozvinutý strom) patří do kategorie adaptabilních datových struktur určených k vyhledávání. Má základní vlastnosti binárních vyhledávacích stromů. V průběhu operací nad touto strukturou se však mění její uspořádání ve prospěch celkového snížení časové složitosti.

Základní mechanismus se nazývá "splay" - rozšíření. Tento mechanismus se začíná od stanoveného uzlu, a postupnými rotacemi způsobuje, že stanovený uzel se stane kořenem stromu, při zachování vyhledávacích relací. Celkovým výsledkem je skutečnost, že často používané položky se hromadí v blízkosti kořene. Na rozdíl od BVS, jehož nejhorší případ pro degenerovaný (lineární) strom má složitost O(N) a je složitost splay stromu pro "k" různých po sobě jdoucích operací O(k*lg(N)). Tato složitost není stanovena tradičním přístupem "worst case", který hledá

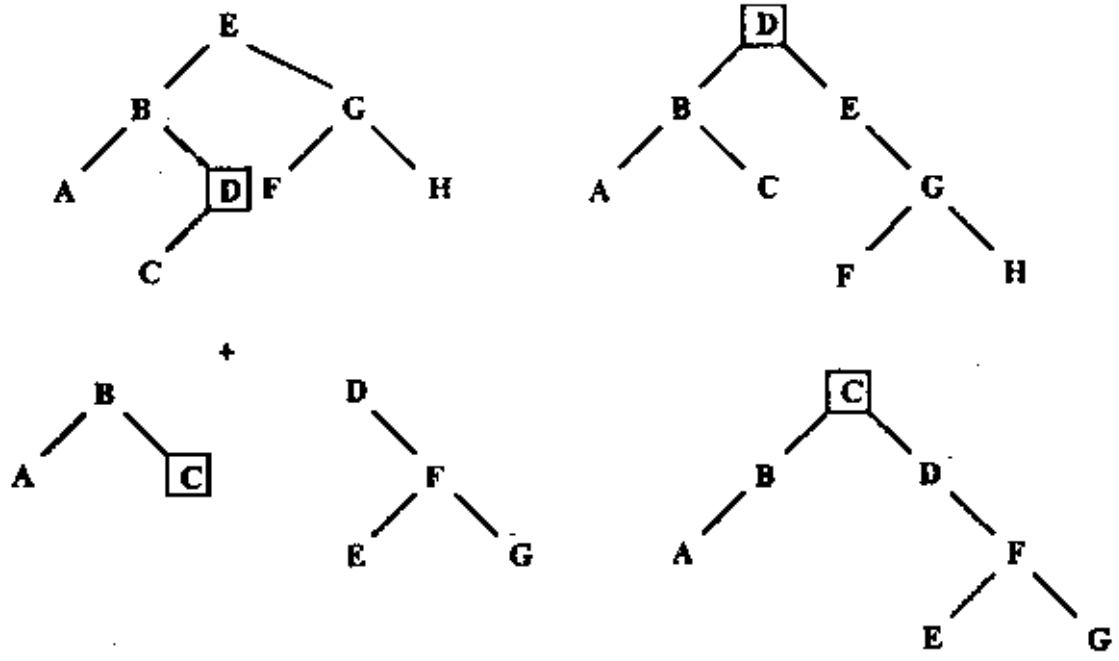
nejnevýhodnější situaci izolované operace, ale metodou "amortizované analýzy" (amortized analysis), která hodnotí celou sekvenci různých operací. Některé z nich jsou delší, některé kratší než lg(N) ale v průměru vychází složitost O(ln(N)).

Mechanismus vyhledání (splay search), pracuje stejně jako u BVS, ale po vyhledání se aplikuje na vyhledaný uzel mechanismus Splay, jehož výsledkem je přesunutí uzlu na místo kořene.

Mechanismus vkládání (splay insert) vloží uzel jako list stejným způsobem jako BVS, ale potom se aplikuje na vložený uzel mechanismus "splay", který opět posune vložený uzel na pozici kořene. Operace "Splay insert" uzlu s klíčem C je uvedena na obr.2.1.

Mechanismus rušení uzlu (splay delete) je poněkud složitější. Uzel, který se má zrušit, se mechanismem splay přesune na pozici kořene. Zrušením kořene získáme 2 podstromy. Mechanismus splay se dále aplikuje na bezprostředního předchůdce a není-li tak následníka zrušeného uzlu (ve smyslu relace uspořádání - v průchodu inorder). Tím se tento uzel dostane do pozice kořene levého podstromu. Podle pravidel vyhledávacího stromu musí být všechny uzly levého podstromu menší než jeho kořen a všechny uzly pravého podstromu větší. Proto musí mít levý podstrom kořen vpravo volný a na toto místo se připojí pravý podstrom. Tento postup má symetrickou - levou verzi. Operace "Splay Delete", rušící uzel D je uvedena na obr.2.2.

Obr.2.1 Splay Insert

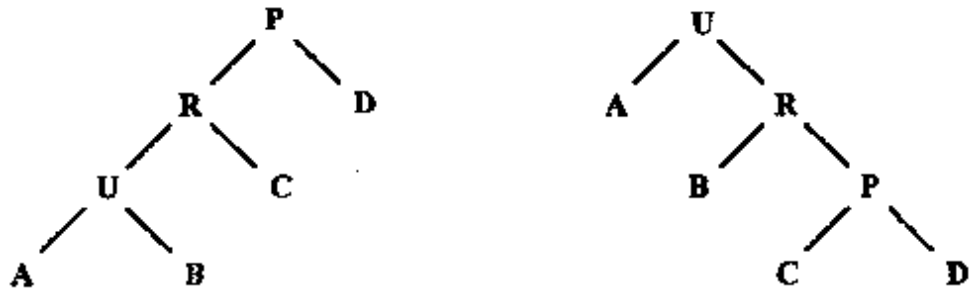


Obr.2.2 Splay Delete

Základem mechanismu "splay" jsou tři typy rotací: Jednoduchá levá rotace "SL" ("zig-left"), dvojitá levá levá rotace "DLL" ("zig-zig- left") a dvojitá pravo-levá rotace "DLR"("zig-zag-left"). Každá rotace má svou symetrickou verzi (SR,DRR a DLR). Pro rotace je třeba znát k danému uzlu (U) rodičovský (R) resp. i prarodičovský (P) uzel.

Obr. 2.3 Jednoduchá rotace "SL" ("zig-left")

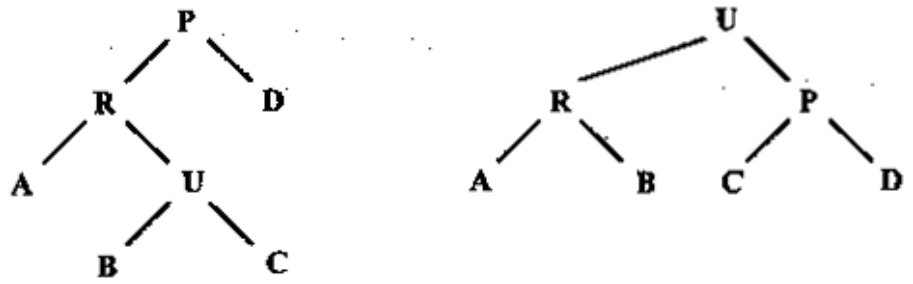
```
procedure SL (var U:TUk);
var B,R: TUk;
begin
  B:=U^.PUk; (* B je pravý syn K *)
  R:=U.ZUk; (* R je rodič K *)
  U^.ZUk:=nil; (* K bude kořen, zpětný ukazatel je nil *)
  U^.PUk:=R;
  if B<>nil then B^.ZUk:=R; (* je-li B neprázdné, má zpětný ukazatel *)
  R^.LUk:=R;
  R^.ZUk:=U;
end;
```



Obr. 2.4 Dvojitá rotace "DLL" ("zig-zig-left").

```
procedure DLL(var U:TUk);
var B,R,P,C,PP:TUk;
begin
  B:=U^.PUk; (* ustavení přístupů k uzlům *)
  R:=U^.ZUk;
  C:=R^.PUk;
  P:=R^.ZUk;
  PP:=P^.ZUk;
```

```
 U^.PUk:=R; R^.ZUk:=U; (* vzájemné připojení R na U *)
 R^.PUk:=P; P^.ZUk:=R; (* vzájemné připojení P na R *)

 R^.LUk:=B; (* připojení B na R *)
 if B<>nil then V^.ZUk:=R;

 P^.LUk:=C; (* připojení C na P *)
 if C<>nil then C^.ZUk:=P;

 if PP<>nil (* napojení uzlu na praprarodiče *)
 then if PP^.PUk = P
     then PP^.PUk:=U
     else PP^.LUk:=U
end;
```



Obr.2.5 Dvojitá rotace "DLR" ("zig-zag-left").

```
procedure DLR(var U:TUk);
var B,C,R,P,PP:TUk;
begin
 B:=U^.LUk; (* ustavení přístupů k uzlům *)
 C:=U^.PUk;
 R:=U^.ZUk;
 P:=R^.ZUk;
 PP:=P^.ZUk;

 U^.LUk:=R; R^.ZUk:=U; (* napojení R na U *)
 U^.PUk:=P; P^.ZUk:=U; (* napojení P na U *)

 R^.PUk:=B; (* napojení B na R *)
 if B<>nil then B^.ZUk:=R;

 P^.LUk:=C; (* napojení C na P *)
 if C<>nil then C^.ZUk:=R;

 if PP<>nil (* napojení U na praprarodiče *)
 then if PP^.PUk=P
     then PP^.PUk:=U
     else PP^.LUk:=U;
end;
```

Jednoduchá rotace SL je nejjednodušší a nastává v případě, kdy otcovský uzel k zadanému uzlu je kořen, jak je vidět z obr 2.3. Pro provedení dvojitých rotací je zapotřebí přístup k rodičovskému i prarodičovskému uzlu.

Dvojitá rotace DLL nastává v případě, že uzel je levým synem rodiče, který je levým synem prarodiče.

Dvojitá rotace DLR nastává v případě, že uzel je pravým synem rodiče, který je levým uzlem prarodiče.

Rotace vyžadují ukazatele k předkům daného uzlu. Tuto situaci lze řešit buď rozšířením každého uzlu o ukazatel na rodičovský uzel pro zpracování zdola-nahoru, nebo použitím zásobníku, v němž se uchovává cesta při zpracování shora-dolů.

Generický mechanismus splay má následující podobu:

```
procedure Splay(K:TKlic);
(* Na strom ukazuje globální proměnná T; Funkce CompKey(K1,K2:TKlic):integer, porovnává dva
   klíče. Vrací -1, je-li první klíč menší, 0 pro rovnost a 1 pro první klíč větší *)

var Comp:integer;
begin
 if T<>nil
 then begin
  Comp:=CompKey(T^.Klic, K);
  while Comp<>0 do begin
   if Comp>0
   then if T^.LUk<>0
      then T:=T^.LUk
      else Halt
   else if T^.PUk<>0
      then T:=T^.PUk
      else Halt;
   end; (* while *)
  (* T nyní ukazuje na uzel s klíčem K, nebo ne předchůdce (následníka) ve smyslu průchodu
     inorder. T se stane kořenem, jeli jeho rodič nilový *)

  while T^.ZUk<>nil do begin
   if T^.ZUk^.ZUk <> nil
   then begin (* dvojitá rotace *)
    if T^.ZUk^.ZUk^.LUk=T^.ZUk
    then (* rodič je levý potomek *)
     if T^.ZUk^.LUk=T
     then (* levý potomek levého potomka *)
      DLL(T); (* Dvojitá rotace DRR - "zig-zig-left" *)
     else (* pravý potomek levého potomka *)
      DRL(T) (* dvojitá rotace DLR - "zig-zag-left" *)
    else (* rodič je pravý potomek *)
     if T^.ZUk^.PUk=T
     then (* pravý potomek pravého potomka *)
```

```
        DRR(T)
       else
        DLR(T)

    else (* jednoduchá rotace *)
     if T^.ZUk^.LUk=T
     then SL(T)
     else SR(T)
   end; (* while *)
  end (* if *)
end; (* procedure *)
```

Operace typické pro vyhledávací stromy - Search, Insert a delete jsou uvedeny v následujících programech.

(* Následující algoritmy předpokládají existenci globální proměnné T, která ukazuje na kořen stromu. Operace používají funkci CompKey(K1,K2):Integer, která vrací hodnotu -1,0 a 1 podle toho zda je první klíč menší, roven nebo větší než druhý. Nil u funkce search je návrat funkce při nenalezení klíče *)

```
function Search(K:TUk):TUk;
begin
  Splay(K);
  (* Pokud byl klíč někde ve stromu, je teď v kořeni *)
  if T^.klic=K
  then Search:=T
  else Search:=0;
end;

procedure Delete(K:TUk);
var
  Levy, Pravy:TUk;
begin
  (* Procedura Splay přivede uzel s daným klíčem do kořene *)
  Splay(K);

  if CompKey(T^.Klic,K)=0
  then begin (* odpoj uzel a zruš ho *)
    Levy:=T^.LUk;
    Pravy:=T^.PUk;
    Dispose(T);
    Levy^.ZUk:=nil;
    Pravy^.ZUk:=nil;

    (* proveď Splay pro levý strom a najdi nový kořen stromu *)
    T:=Levy;
    Splay(T);

    (* napoj pravý strom na kořen levého *)
    T^.PUk:=Pravy;
  end;
end;
```

```
procedure Insert(K:TUk);
(* Tato verze programu popisuje princip ale s ohledem na dvojí průchod
stromem je tato implementace nevhodná *)
begin
  NajdiAPripojUzel(K);
  (* procedura připojí nový uzel jako koncový uzel v případě nenalezení
  klíče ve stromu *)

  Splay(K);
  (* procedura Splay přesune vložený uzel do kořene *)
end;
```

4. Hodnocení SPLAY stromů

Splay stromy představují jeden z příkladů adaptibilních datových struktur, jejichž vnitřní uspořádání se mění vlivem jako vedlejší jev operací nad těmito strukturami. Mají dobrou tendenci vyvažovat stromovou strukturu a svou vlastností přibližovat často vyhledávané klíče kořeni se podobají adaptibilní lineární struktuře pro sekvenční vyhledávání, v níž se každý vyhledaný uzel vymění se svým levým předchůdcem. I ve stromové podobě si algoritmus zachovává jednoduchost a průhlednost.

Literatura :

[Hon] Honzík,J: Vyhledávání v binárních stromech, sborník Programování '85, DT ČSVTS Ostrava 1985

[Sed] Sedgewick,R.: Algorithms, Addison-Wesley, 1988

[Sle] Sleator,D.D., Tarjan,R.E.: Self-Adjusting Binary Search Tree, Journal of the ACM, July, 1985

[Cla] Clark,D.: Splay Tree, Dr.Dobbs Journal, #195, December 1992

[Cor] Cormen,T.H.,Leiserson,C.E.,Rivest,R.L., Introduction to Algorithms, The MIT Press, McGraw-Hill Book Comp.,1990

Autor :

Doc.Ing.Jan M Honzík,CSc
Ústav informatiky a výpočetní techniky
FEI VUT v Brně
Božetěchova 2, 612 66 Brno 12
tel.: (05)-7275217
fax : (05)-41211141
e_mail:honzik@dcse.fee.vutbr.cs